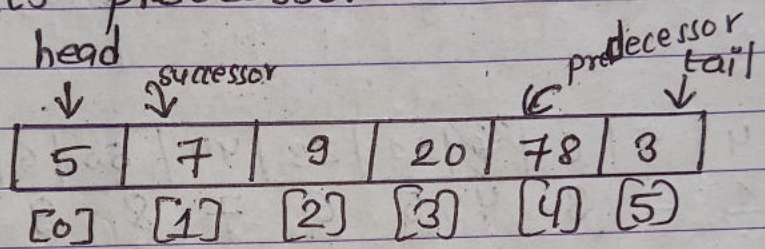
Unit-5
List

#List:

A list is an ordered set consisting of a number of elements to which addition and deletion can be made. The first element of a list is called the head of the list and last is called tail of the list. The element next to the head of the list is called its successor. The previous element to the tail is called its predecessor.

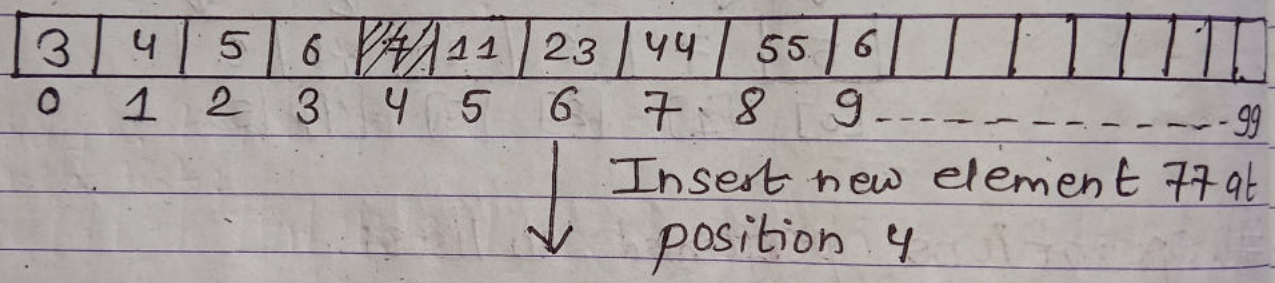


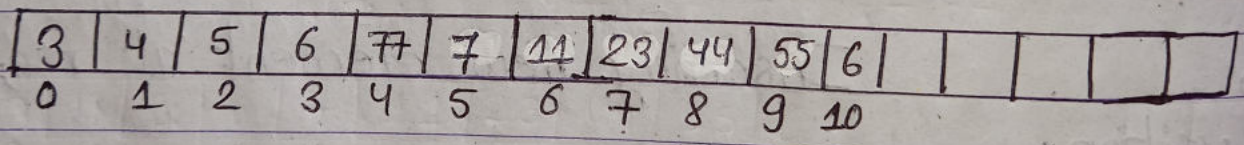
 # Operations performed on lists:

- Inserting new element at require position.
- Deleting of an existing element from given list.
- Modification of an existing element from given list.
- Traversing of an existing list.
- Merging of any two existing list.

(1) Inserting new element at require position:

This operation is used to insert new element at required position from 0 to n^{th} index of given list where 0^{th} index is the first index of given list and n^{th} index is the $(last+1)^{th}$ index of given predefined list of size 'n'. Here, we need to shift every element one position right from last element to specified positions element.





Algorithm:

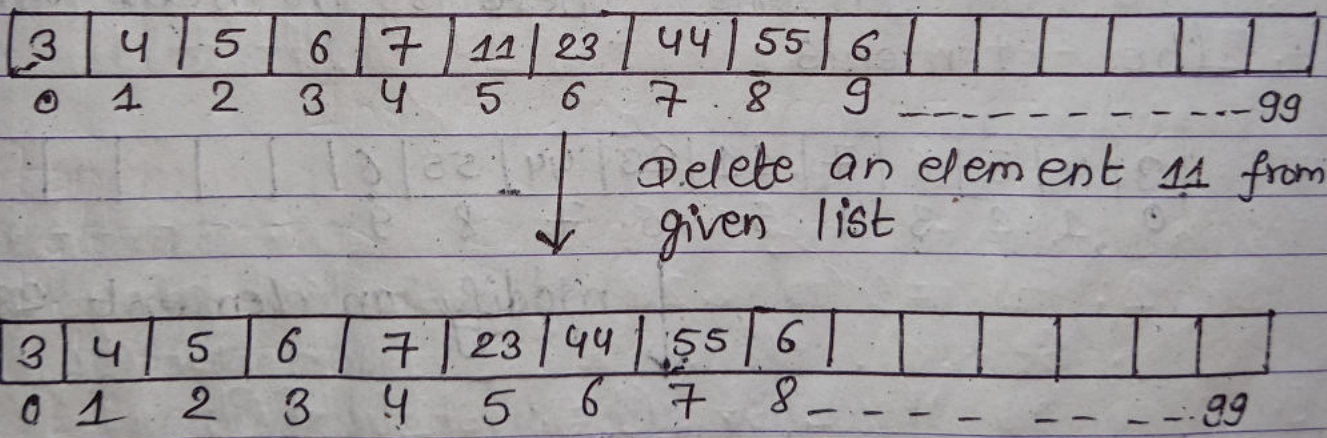
Step 1: Start
Step 2: Read position of element to be inserted say it be 'pos'.
Step 3: Read element to be inserted say it be 'el'.
Step 4: Swap all elements one position right from last index to (pos-1).

Step 5: Now location for new element is free, so set new element at that location at list[pos] = el;

Step 6: Increment size of an array by 1 i.e.
n = n+1 or n++1

(2) Deleting of an existing element from given list:

In this we need to shift all elements one position left from index of deleted element to the index of last element and finally decrease size of given list by 1.

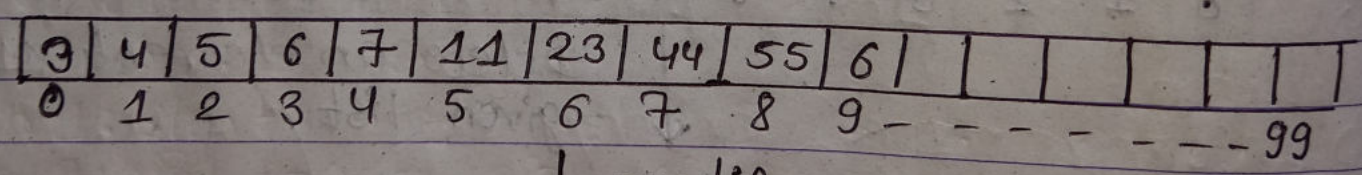


Algorithm:

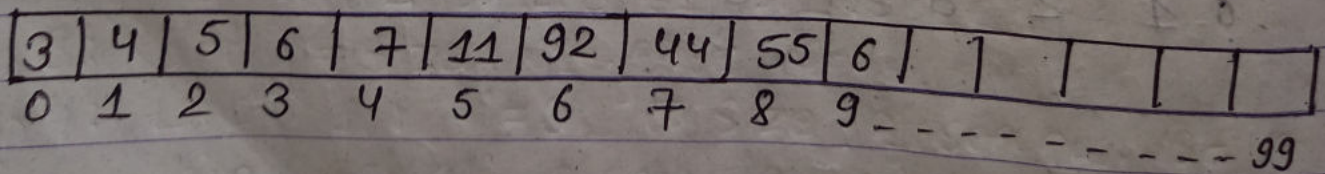
Step 1: Start
Step 2: Read position of element to be deleted say it be 'pos'.
Step 3: Swap all elements one position left from specified position to last element.
Step 4: Decrement size of a list by 1 i.e. n=n-1.
Step 5: stop

(3) Modification of an existing element from given list:

In this there is no need to shift the elements



modify an element 23 by 92



Algorithm:

Step 1: Start
Step 2: Read position of an element to be updated say it 'pos'.
Step 3: Read element to be updated say it be 'el'.
Step 4: Set element at given position as list [Pos] = el;
Step 5: Stop

(4) Traversing of an existing list:

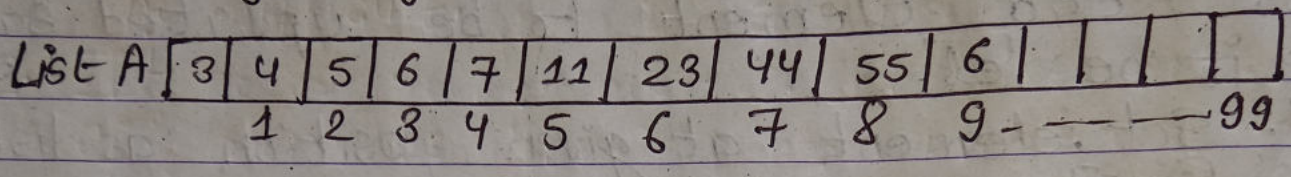
The operation is used to display the elements from first index to last index of given list. Here, we donot need to shift of elements.

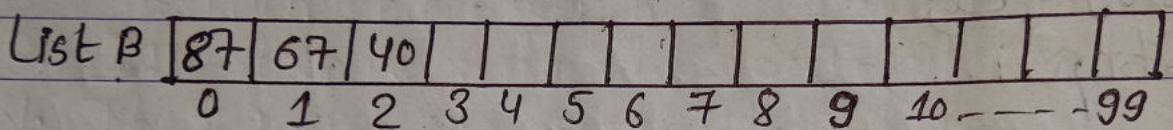
Algorithm:

Step 1: Start
Step 2: Loop for i=0 to (list_size - 1)
Display list[i]
Increment i by one as i=i+1
Step 3: stop

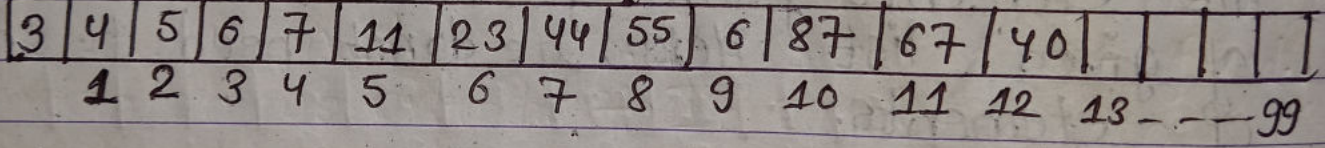
(5) Merging of any two existing list:

The operation is used to append all the elements of one list to the end of another list. Here, we do not need to shift of elements.





↓ Merging of given two lists into single list.



Algorithm:

Step 1: start

Step 2: For i=0 to size of first last minus one
Set all elements of first list to new list as New_list [i] = first_list[i];

Step 3: Set j = size of first list

Step 4: Loop for i=0 to size of second list minus are set all elements of second list to new list as,
New_list [j] = second_list [i]

Increment j by one as, j=j+1

Step 5: Increment size of new list by size of first list plus size of second list.

Step 6: Stop.

Linked List:

A linked list is an ordered collection of finite homogeneous data elements called nodes where the linear ~~dat~~ order is maintained by means of link or pointer. A linked list consists of nodes of data which are connected with each other. Every node consists of two fields data (info) and the link (next). The nodes are created dynamically.

- Info: The actual element to be stored in the list. It is also called data field.
- Link: one or two links that point to next and previous node in the list. It is also called next or pointer field.

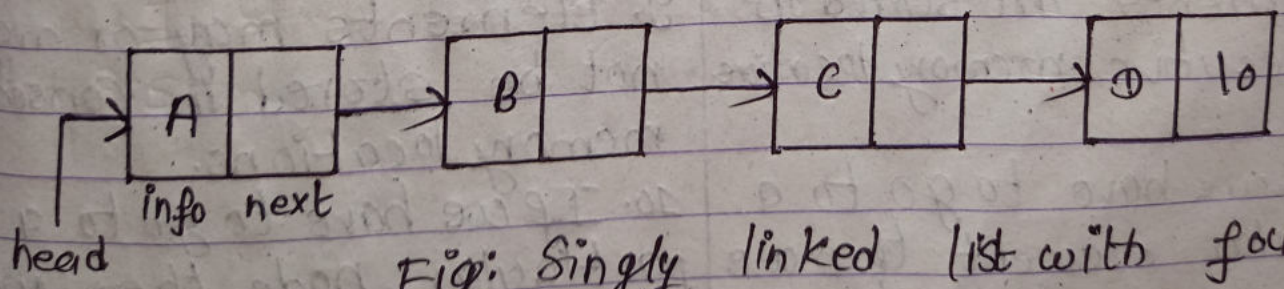


Fig: Singly linked list with four nodes.

Differences between linear array and linked list

Linear Array	Linked list
1. Insertions and deletions ~~can be done easily~~ are difficult.	1. Insertions and deletions can be done easily.
2. It needs movements of elements for insertion and deletion.	2. It does not need movement of nodes for insertion & deletion.
3. In it space is wasted.	3. In it space is not wasted.
4. It is more expensive.	4. It is less expensive.
5. It requires less space as only information is stored.	5. It requires more space as pointers are also stored along with information.
6. Its size is fixed.	6. Its size is not fixed.
7. It cannot be extended or reduced according to requirements.	7. It can be exten
8. Same amount of time is required to access each element.	8. Different amount of time is required to access each element.
9. Elements are stored in consecutive memory locations	9. Elements may or may not be stored in consecutive memory locations.
10. If we have to go to a particular element then we can reach their directly.	10. If we have to go to a particular node then we have to go through all those nodes that come before that node.

Uses of linked list in real world:

(1) Image viewer :
Previous and next images are linked, hence can be accessed by next and previous button.

(2) Previous and next page in web browser:
We can access previous and next URL searched in web browser by pressing back & next button. Since, they are linked as linked list.

(3) Music player : Songs in music player are ~~in~~ linked to previous & next song. We can play songs either from starting or ~~reading~~ ending of the list.

Applications of linked list:

(1) Implementation of stack and queue.
(2) Implementation of graph.
(3) Dynamic memory allocation.
(4) Maintaing directory of names.
(5) Performing arithmetic operations on long integers.

Advantage of linked list:

(1) Linked list is an example of dynamic data structure. They can grow and shrink during the execution of program.

(2) Efficient memory utilization. The allocation of memory depends upon the user i.e. no need to pre-allocate memory.

(3) Insertion and deletion can be easily performed.

(4) Linear data structure such as stack, queue can be easily implemented using linked list.

(5) Faster access time, can be expanded in constant time without memory overhead.

Types of linked list:

(1) Singly linked list
(2) Doubly linked list
(3) Circularly linked list
(4) Circularly doubly linked list.

(1) Singly linked list:

A singly linked list is a dynamic data structure which may grow or shrink. In this type of linked list, each node contains two fields: One is info field or data field which is used to store the data items and another is link or next field that is used to point the next node in the list. The last node has a NULL pointer. The address of first node can be contained by an external pointer called 'First'.

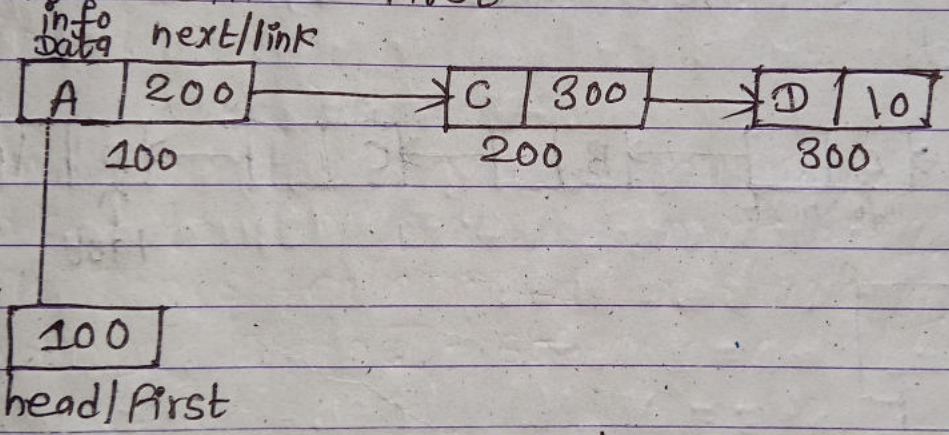


Fig: Singly linked list

* Algorithm to insert a node at the beginning:

Let first and last are the pointers to first node and last node in the current list respectively.

Step 1: Start
Step 2: Create a new node using malloc() function as,

Newnode = (Nodetype *) malloc (size of Nodetype);

Step 3: Read data item to be inserted say 'el' element
Step 4: Assign data to the info field of new node

Newnode·info = el;

Step 5: Set next of newnode to first

Newnode·next = first;

Step 6: Set the first pointer to the newnode

first = Newnode;

Step 7: End

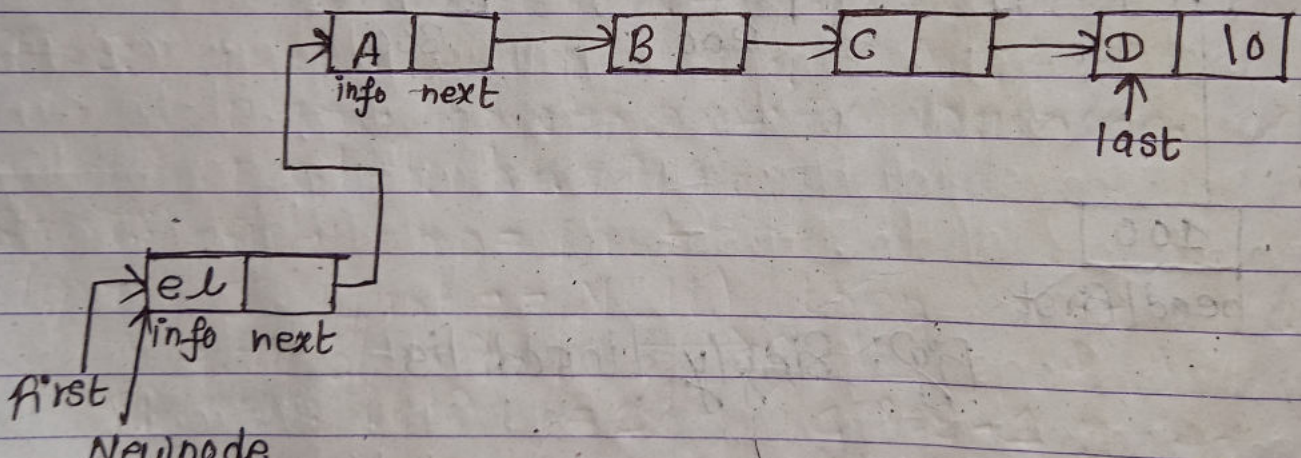
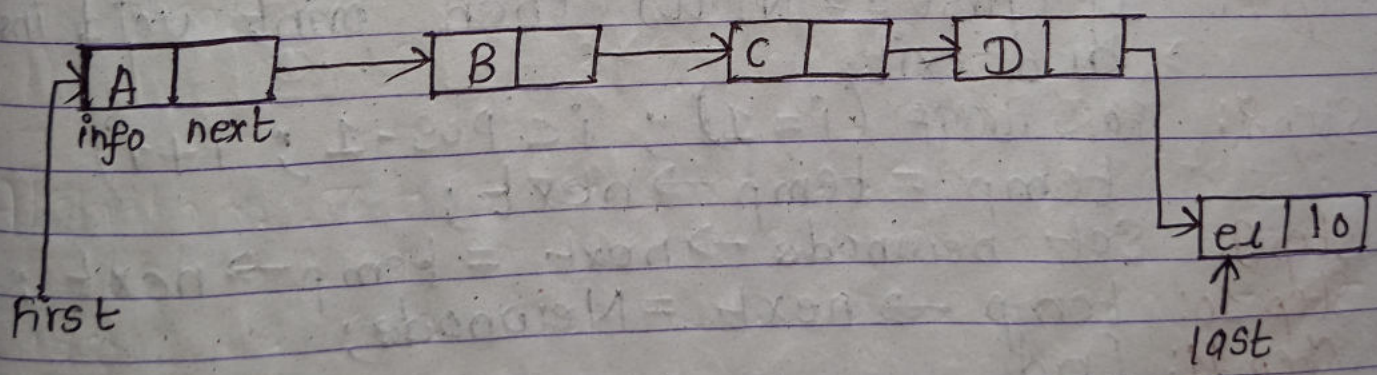


Fig: Algorithm ~~for~~ to Insert a node at the beginning

* Algorithm to insert a node at the end:

Step 1: Start
Step 2: Create a new node malloc() function as,
Newnode = (Nodetype *) malloc (size of Nodetype);
Step 3: Read data item to be inserted say 'el' element.
Step 4: Assign data to the info field of newnode.
Newnode → info = el;
Step 5: Set next of newnode to NULL
Newnode → next = NULL;
Step 6: If (first == NULL) then, set first = last = newnode and exit.
Step 7: Else set last → next = Newnode;
Set last = Newnode;
Step 8: End



* Algorithm to insert a node at the specified point:

Let first and last are the pointer to first node and last node in the current list respectively.

Step 1: Start
Step 2: Create a new node using malloc() function as,

Newnode = (Nodetype *) malloc (Size of Nodetype);

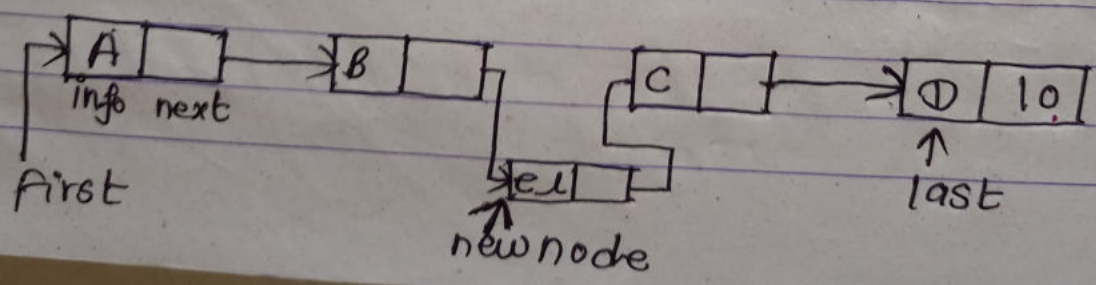
Step 3: Read data item to be inserted say 'el' element.
Step 4: Assign data to the info field of newnode

Newnode → info = el;

Step 5: Enter position of a node at which we want to insert a newnode. Let this position be 'pos'.
Step 6: Set temp = first
Step 7: If (first == NULL) then print void insertion and exit.
Step 8: FOR (i = 1 ; i < pos-1 ; i++)

temp = temp → next;

Step 9: Set newnode → next = temp → next;
Step 10: temp → next = Newnode;
Step 11: End



* Algorithm to delete the first node of singly linked list:

Let first and last are the pointer to first and last node in the current list respectively

Step 1: Start

Step 2: If (first == NULL) then
print (void deletion) and exit

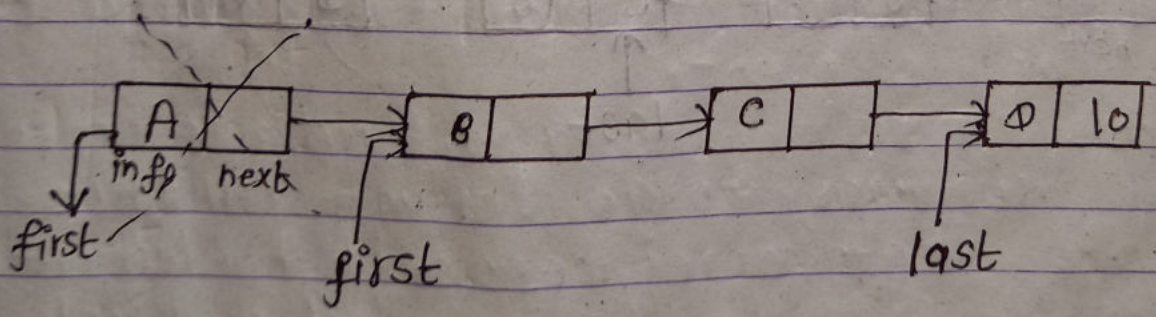
Step 3: Else If (first == last) then
print (deleted item as first.info);
first = last = NULL

Step 4: Store the address of first node in a temporary variable 'temp'.
temp = first;

Step 5: Set first to next of first
Set, first = first → next

Step 6: Free the memory reserved by memory
free(temp);

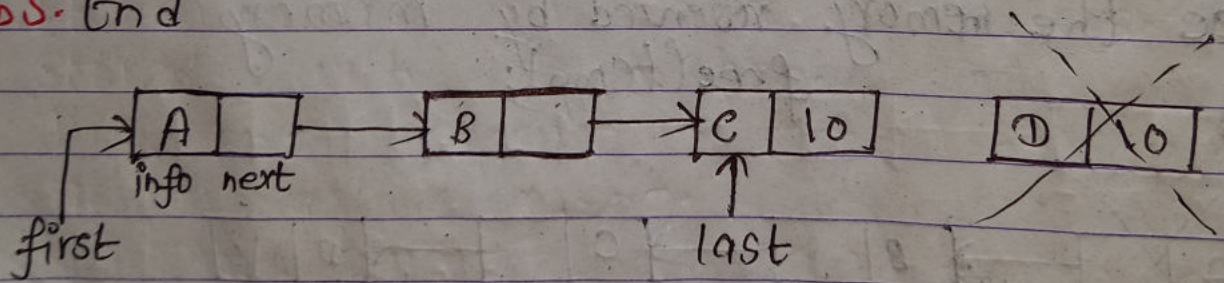
Step 7: End



* Algorithm to delete the last node of singly linked list:

Let first and last are the pointer to first and last node in the current list resp.

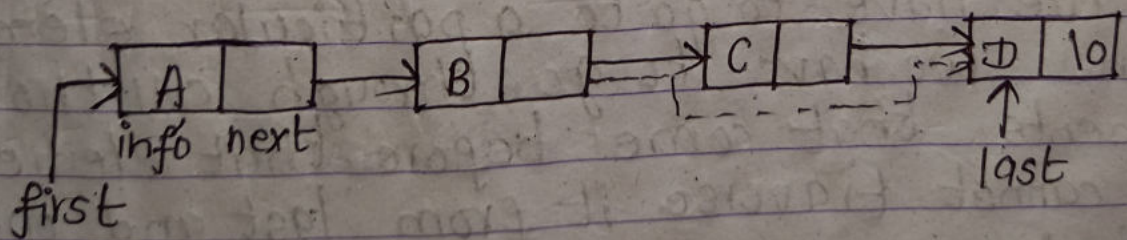
Step1: Start
Step2: If (first == NULL) then
print (void deletion) and exit
Step3: Else if (first == last) then
print (deleted item as first.info);
Step4: Set, first = last = NULL.
Step5: Else temp = first
while (temp → next != last)
set temp = temp → next;
set temp → next = NULL;
set last = temp;
Step5: End



* Algorithm to delete a node at the specified position singly linked list:

Let first and last are the pointer to first and last node in the current list respectively.

Step1: Start
Step2: Read position of a node which is to be deleted. Let it be pos.
Step3: If (first == NULL) then
print (void deletion) and exit
Step4: Otherwise enter position of a node at which we want to delete a new node. Let it be LOC.
Step5: Set temp = first
Step6: for (i=1; i< pos-1; i++)
Set: temp = temp → next;
Step7: print deleted item is temp · next · info
Step8: Set LOC = temp → next;
Step9: Set temp → next = loc → next
Step10: End



Advantages of singly linked list:

(1) Insertions and Deletions can be done easily.
(2) It does not movement of elements for insertion and deletion.
(3) Its space is not wasted as we can get space according to our requirements.
(4) Its size is not fixed.
(5) It can be extended or reduced according to requirements.
(6) Elements may or may not be stored in consecutive memory available ; even then we can store the data in computer.
(7) It is less expensive.

Disadvantages of singly linked list:

(1) It requires more space as pointers are also stored with information.
(2) Different amount of time is required to access each element.
(3) If we have to go to a particular element then we have to go through all those elements that come before that element.
(4) We cannot traverse it from last and only from beginning.
(5) It is not easy to sort the elements stored in the linear linked list.

The Linked list act as ADT:

A linked list of n-nodes with n-elements of type T is a sequence of elements of T together with the operations:

- Create(): Create or make a node
- Insert(x): Insert x to linked list
- Delete (): If linked list is not empty then delete given node.
- Traverse(): Display all of the nodes of given linked list.
- IsEmpty(): Determine whether linked list is empty or not. Return true if it is empty return false otherwise.
- find() or search(): Find out given node from linked list.
- Count(): Count number of nodes of given linked list.
- Free(): release memory space of given node of linked list.

P.T.O. →

Circular Linked List:

A circular linked list is a list where the link field of last node points to the first node of the list. Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node. The main advantage of circular linked list is that it requires minimum time to traverse the nodes which are already traversed, without moving to starting node.

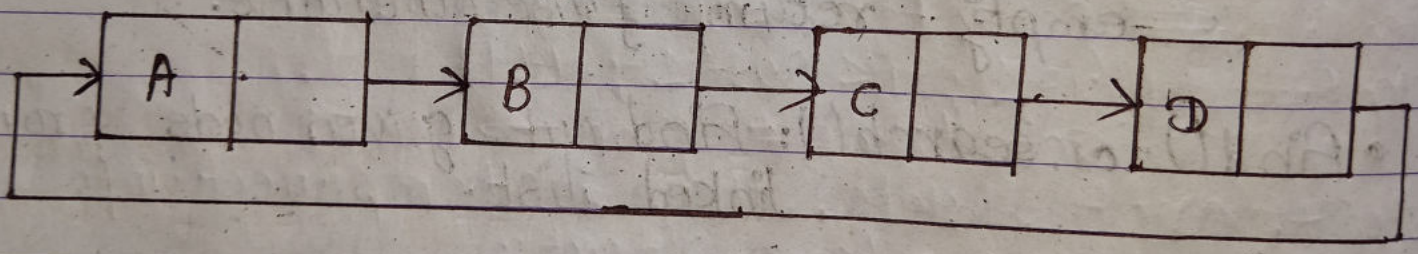


Fig: Circular linked list with four nodes

Advantages of circular linked list:

(1) If we are at a node, then we can go to any node. But in linear linked list it is not possible to go to previous node.

(2) It saves time when we have to go to the first node from the last one. It can be done

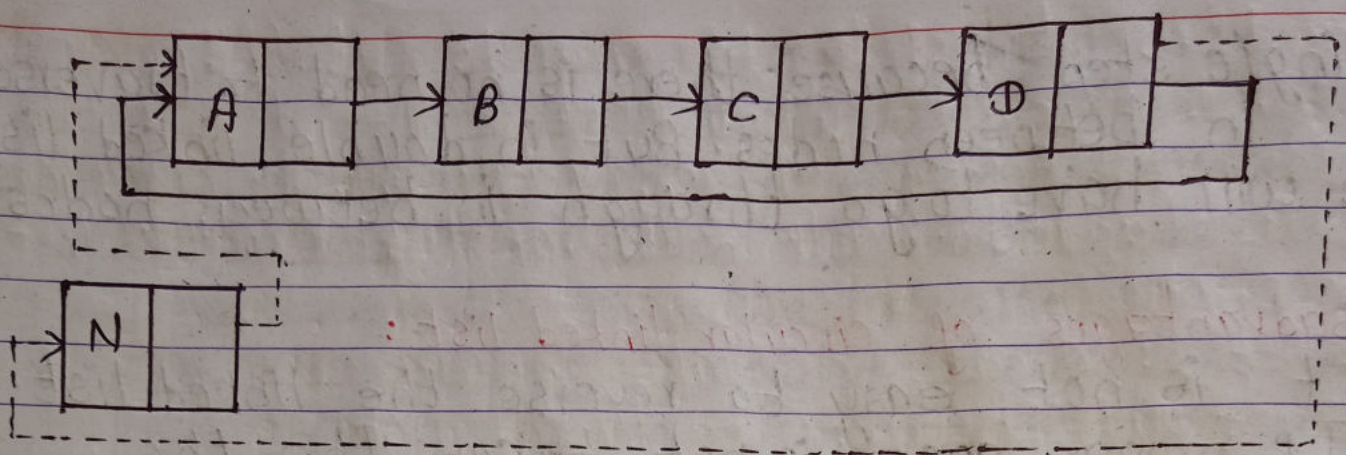
in single step because there is no need to traverse the in between nodes. But in double linked list, we will have to go through in between nodes.

Disadvantages of circular linked list:

1. It is not easy to reverse the linked list.
2. If proper case is not taken, then the problem of infinite loop can occur.
3. If we at a node and go back to the previous node, then we cannot do it in single step.

* Algorithm to insert a node at the beginning of a circular linked list:

Step 1: Start
Step 2: Create a new node by using malloc function as,
Newnode = (NodeType*) malloc (size of (NodeType));
Step 3: Read data item to be inserted say it be 'el'.
Step 4: Set Newnode → info = el
Step 5: if first == null then
Set, Newnode → next = Newnode
Set, first = Newnode
Set, last = Newnode
Step 6: else
Set, Newnode → next = start
Set, first = Newnode
Set, last → next = Newnode
Step 7: End



* Algorithm to insert a node at the end of a circular linked list:

Step 1: Start

Step 2: Create a new node by using malloc function as,
Newnode = (NodeType*) malloc (sizeof (NodeType));

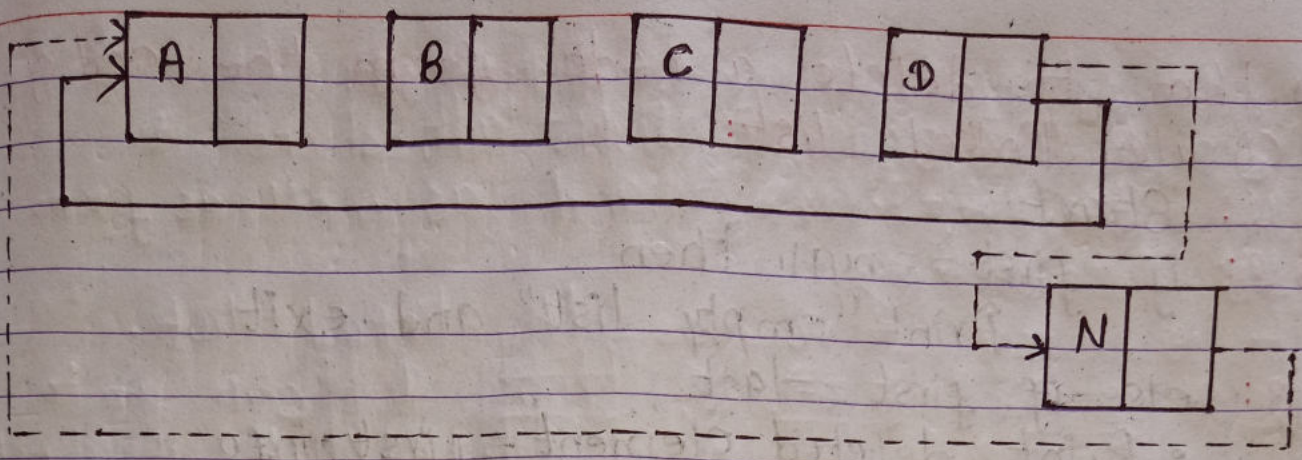
Step 3: Read data item to be inserted say it be 'el'.

Step 4: Set Newnode → info = el

Step 5: If start == null then
Set, Newnode → next = Newnode
set, start = Newnode
Set, last = Newnode

Step 6: else
Set, last.next = Newnode
Set, last = Newnode
Set, last.next = start

Step 7: End



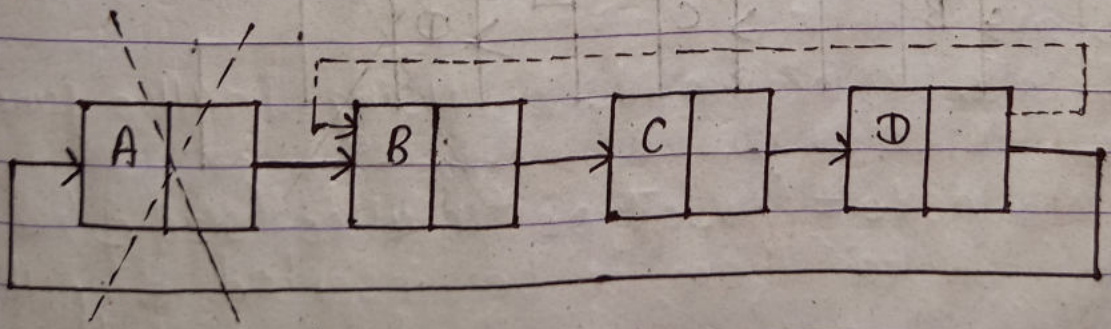
* Algorithm to delete a node from the beginning of a circular linked list:

Step 1: Start

Step 2: if first == null then
Print "Empty list" and exit

Step 3: else
Print the deleted element = first → info
Set, first = first → next
Set, last → next = first;

Step 4: End



P.T.O. →

*Algorithm to delete a node from the end of a circular linked list:

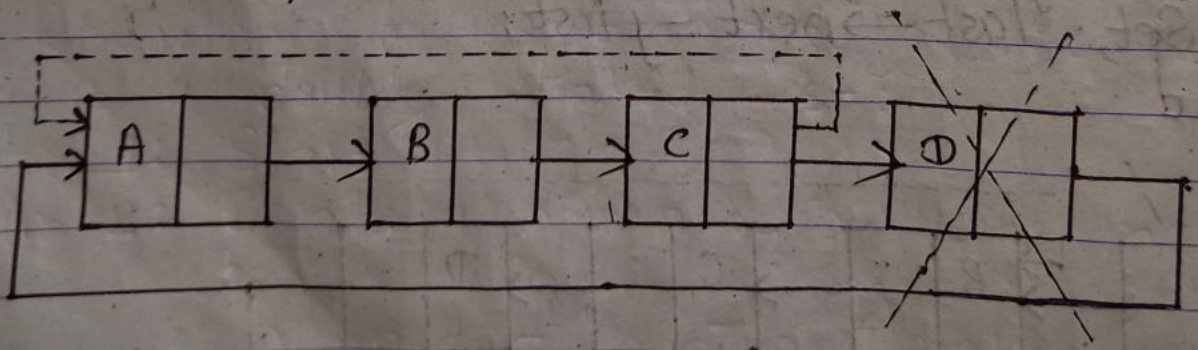
Step 1: Start

Step 2: if first == null then
Print "empty list" and exit

Step 3: else if first == last
- Print deleted element = first.info
- Set, first = last = null

Step 4: else
Set, temp = first
while (temp → next != last)
Set, temp = temp → next
End while
Print the deleted element = last → info
Set, last = temp
Set, last → next = first

Step 5: Stop



Doubly Linked List:

A linked list in which all nodes are linked together by multiple numbers of links i.e. ~~contai~~ each node contains three fields (two pointer fields and one data field) rather is called doubly linked list. It provides bidirectional traversal. ~~Doubly circular linked list can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.~~

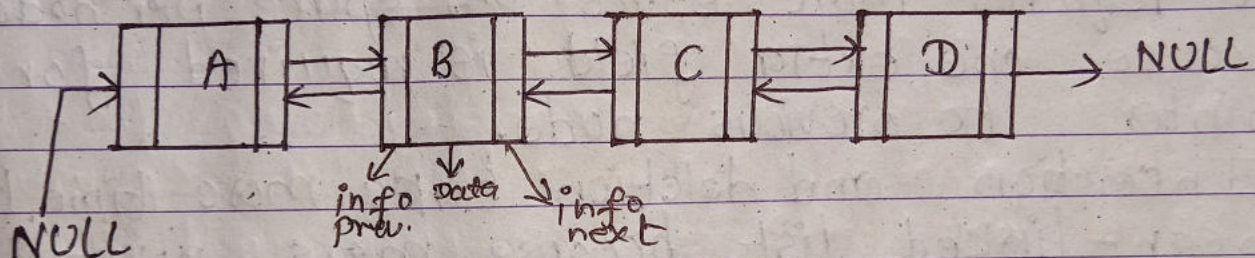


fig: Doubly linked list with three nodes

~~The above diagram represents the basic structure of Doubly Circular linked list.~~ In doubly linked list we cannot access predecessor node from current node. This problem is overcomed by doubly linked list.

Advantages of doubly linked list:

(1) We can traverse in both directions i.e. from starting to end and as well as from end to starting.
(2) It is easy to reverse the linked list.
(3) If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Disadvantages of doubly linked list:

(1) It requires more space per space per node because one extra field is required for pointer to previous node.
(2) Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list

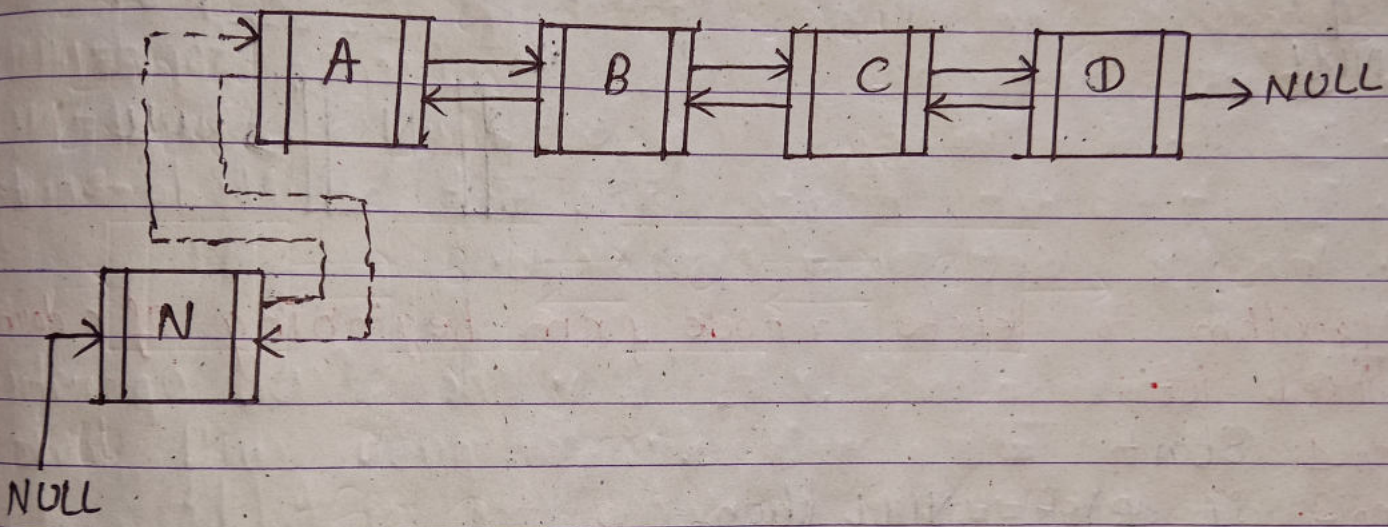
* Algorithm to insert a node at the beginning of a doubly linked list:

Step 1: Start
Step 2: Create a new node by using malloc function as,

Newnode = (NodeType *)malloc (size of (NodeType))

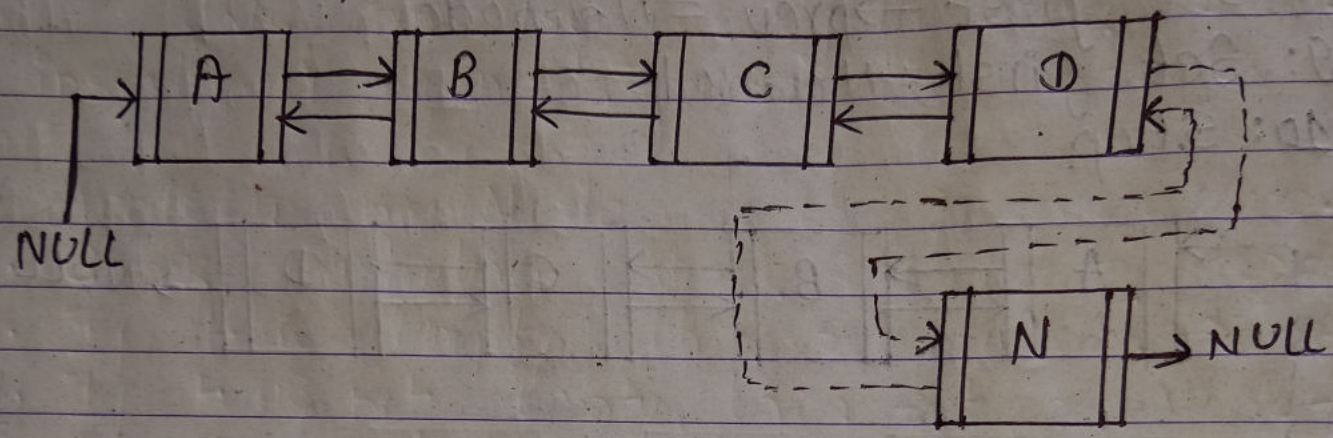
Step 3: Read data item to be inserted say it be 'el'.
Step 4: Set Newnode → info = el
Step 5: Set Newnode → prev = Newnode → next = null

Step 6: If first==null then
Set, first = last = Newnode
Otherwise,
Step 7: Set Newnode → next = first
Step 8: Set first → prev = Newnode
Step 9: Set first = Newnode
Step 10: Stop



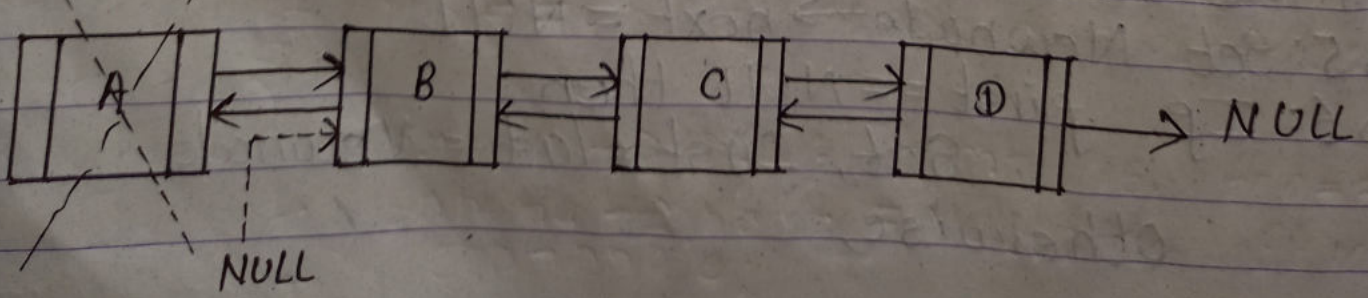
* Algorithm to insert a node at the end of a doubly linked list:
Step 1: Start
Step 2: Create a new node by using malloc function as,
Newnode = (NodeType*)malloc(sizeof(NodeType))
Step 3: Read data item to be inserted say it be 'el'.
Step 4: Set Newnode → info = el
Step 5: Set Newnode → next = NULL
Step 6: If first == NULL then
Set, first = last = Newnode
Otherwise,

Step 7: Set last → next = Newnode
Step 8: Set Newnode → prev = last
Step 9: Set last = Newnode
Step 10: Stop



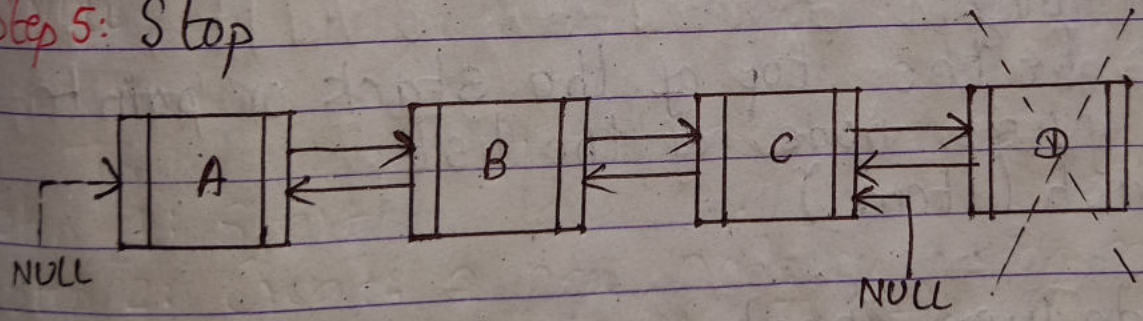
* Algorithm to delete a node from beginning of a doubly linked list:

Step 1: Start
Step 2: if first == NULL then
Print "empty list" and exit
Step 3: else
set, temp = first
Set, first = first → next
Set, first → prev = null
Free (temp)
Step 4: Stop



* Algorithm to delete a node from end of a doubly linked list:

Step 1: Start
Step 2: if first == NULL then
        Print "empty list" and exit
Step 3: else if (first == last) then
        Set, first = last = NULL
Step 4: else
        Set, temp = first;
        while (temp → next != last)
            temp = temp → next
        end while
        Set temp → next = null
        Set, last = temp
Step 5: Stop



Linked List Implementation of Stack (OR Dynamic Implementation of Stack):

* PUSH Operation:

We can use the following steps to insert a newnode into the stack.

1. Create a newNode with given value.
2. Check whether stack is Empty (top == NULL)
3. If it is Empty, then set newNode → next = NULL
4. If it is not Empty, then set newNode → next = top.
5. Finally, set top = newNode.

The time complexity for the Push operation is O(1). The method for push will look like following:

Let, *top be the top of the stack or pointer to the first node of the list.

```
void push (item)
{
    NodeType *nnode;
    int data;
    nnode = (NodeType *) malloc (sizeof (NodeType));
    if (top == 0)
    {
        nnode → info = item;
        nnode → next = NULL;
        top = nnode;
    }
```

```
else
{
    nnode->info = item;
    nnode->next = top;
}
}
```

*POP Operation:

To pop an item from a linked stack, we just have to reverse the operation. Let *top be the top of the stack or pointer to the first node of the list. We can use the following steps to delete a node from the stack:

1. Start
2. Check whether stack is empty i.e (top==NULL)
3. If it is empty, then display "stack is empty !!! Deletion is not possible !!!" & terminate the function.
4. It is not empty, then define a Node pointer 'temp' & set it to 'top'.
5. Then set 'top = top → next'
6. Finally, delete 'temp' as free (temp)
7. Stop

The time complexity for pop operation is O(1). The method for pop ~~operation~~ ~~is~~ will look like the following:

```
void pop()
{
    Node Type *temp;
    If (top==0)
    {
      printf ("Stack contain no elements :\n");
      return;
    }
else
{
    temp= top;
    top= top -> next;
    printf ("\n deleted item is :%d\t", temp
    ->info);
    free (temp);
}
}
```

* Display elements of Stack:

We can use the following steps to display the elements (nodes) of a stack:

1. Start
2. Check whether stack is empty as, (top==NULL)
3. If it is empty, then display 'Stack is

Empty !!!' and terminate the function.
4. If it is not Empty, then define a node pointer 'temp' and initialize with top.
5. Display 'temp → data' and move it to the next node. Repeat the same until temp reaches to the first node in the stack (temp → next != NULL).
6. Stop

Linked List Implementation of Linear Queue

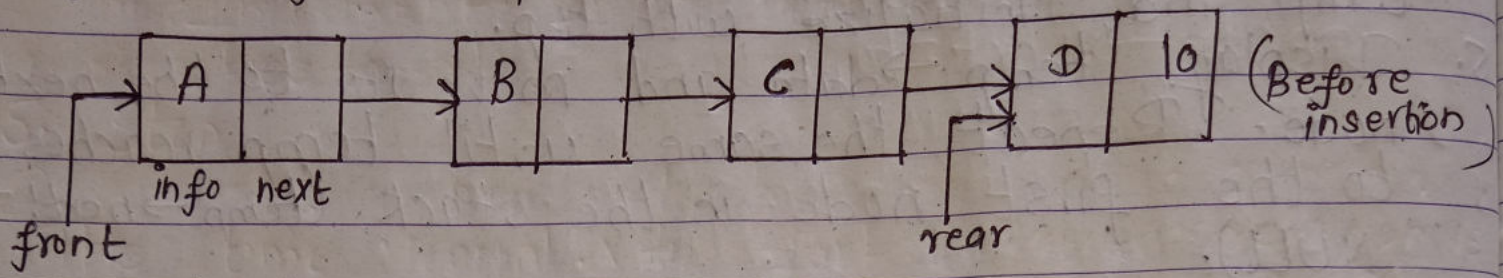
* Algorithm:

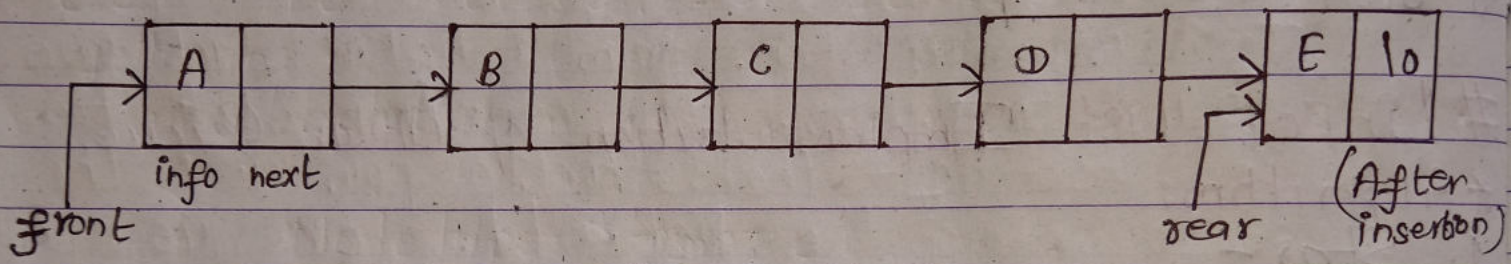
1. Start
2. Create a new node say newNode.
3. Set the value to be inserted into info field of newNode.
4. If the Queue is empty, then set both front and rear to point to newNode and set next field of newNode to NULL.
5. If the Queue is not empty, then set next of rear to the newNode and the rear to point to the new node.
6. Stop

* Insert Function:

Let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part

and deletion of node from the linked list done from front part.



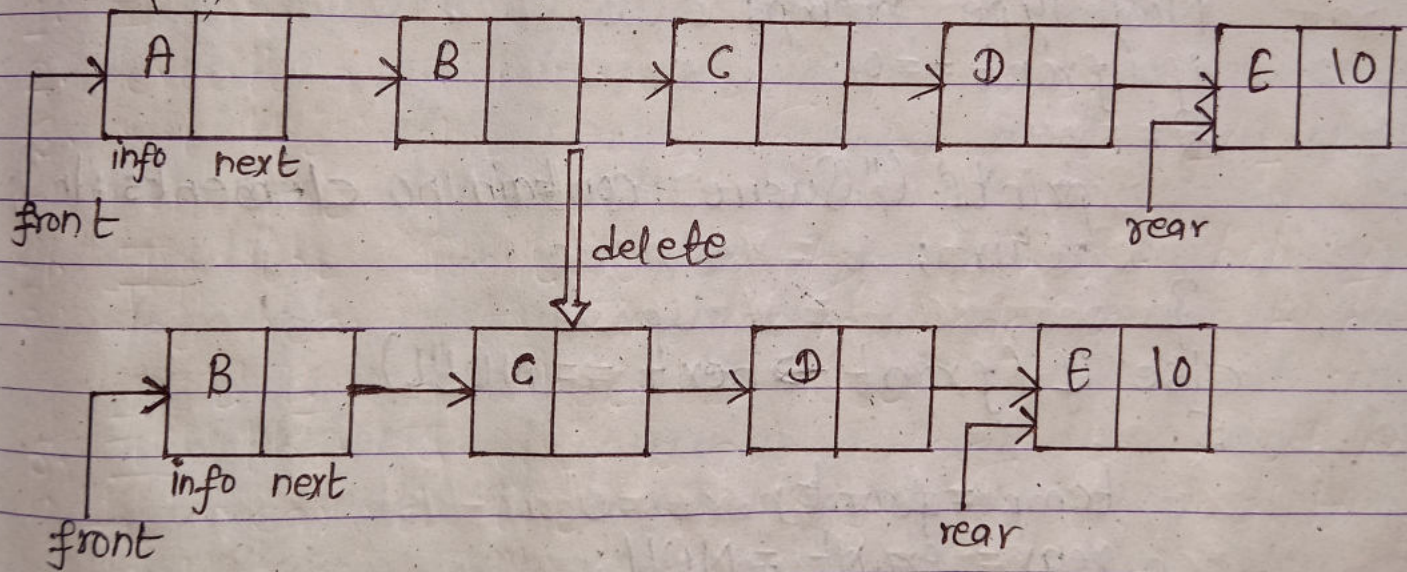



```
Void insert (int item)
{
    NodeType *nnode;
    nnode = (NodeType *)malloc (Size of (NodeType));
    if (rear == 0)
    {
        nnode → info = item;
        nnode → next = NULL;
        rear = front = nnode;
    }
    else
    {
        nnode → info = item;
        nnode → next = NULL;
```

```
    rear -> next = nnode;
    rear = nnode;
  }
}
```

*Delete function:

Let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.



*Algorithm:

1. Start
2. If the Queue is empty, terminate the method.
3. If the next node of front is null then
   Set temp = front
   Set rear = front = NULL
   free(temp)

4. Otherwise
   Set, temp = front
   Increment front to point to next node
   Free(temp)
5. Stop

The time complexity for Dequeue operation is O(1). The method for Dequeue will be like following.

```
void delet()
{
    NodeType *temp;
    if (front == 0)
    {
        printf ("Queue contain no elements: \n");
        return;
    }
    else if (front -> next == NULL)
    {
        temp = front;
        rear = front = NULL;
        printf ("\n Deleted item is %d \n", temp -> info);
        free (temp);
    }
    else
```

```
    {
        temp = front;
        front = front -> next;
        printf("\n Deleted item is %d \n", temp->info);
        free(temp);
    }
}
```

Linked List Implementation of Circular Queue:

By using circular linked list we can easily simulate circular queue. Inserting node at beginning of circular linked list act as enqueue operation of circular queue. Similarly deleting last node of circular linked list act as dequeue operation. In circular queue we can insert element from rear end and delete element from front end. Like this linked list implementation of circular queue we can insert new node at beginning of circular linked list i.e. from rear parts and deleted from end of circular linked list i.e. from front part of circular linked list.

P.T.O. →

* Insertion Algorithm:

Inserting node at beginning of ciraular linked list act as enqueue operation.

1. Start

2. Create a new node by using malloc function as,
   Newnode = (struct linked cQueue*) malloc (size of (struct linked cQueue));

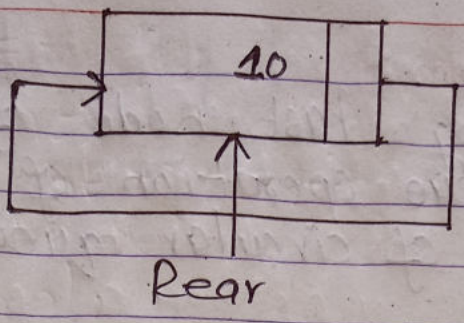
3. Read data item to be inserted say it be 'el'.

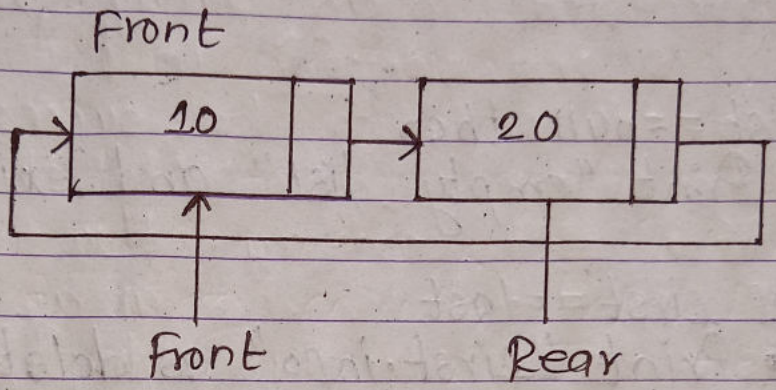
4. Set Newnode → info = el

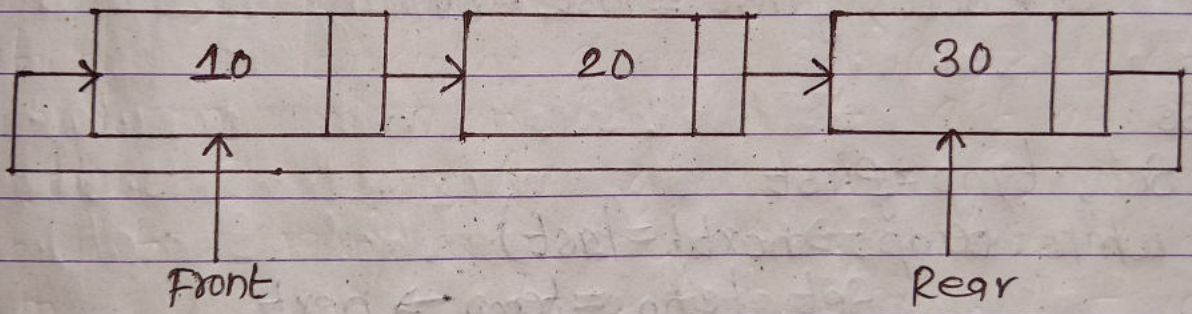
5. if first == null then
   Set, Newnode → next = Newnode
   Set, first = Newnode
   Set, last = Newnode

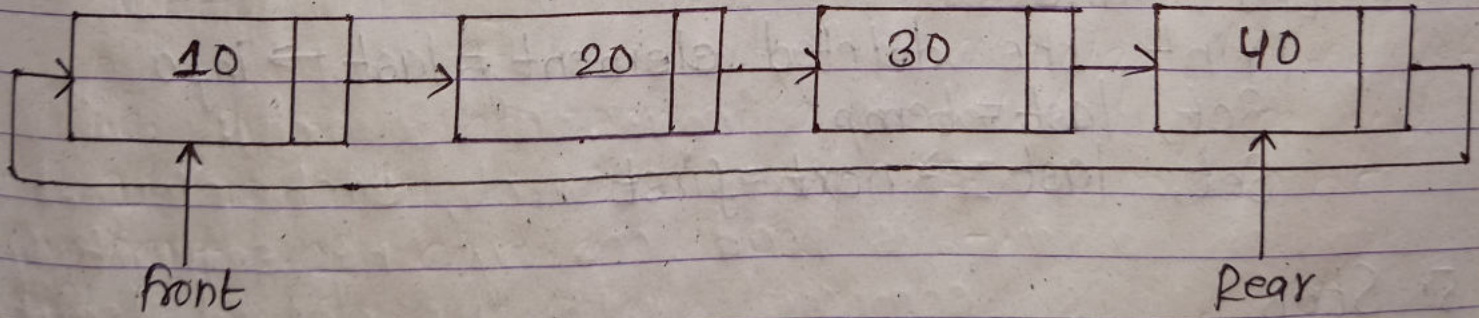
6. else
   Set, Newnode → next = start
   Set, first = Newnode
   Set, last → next = Newnode

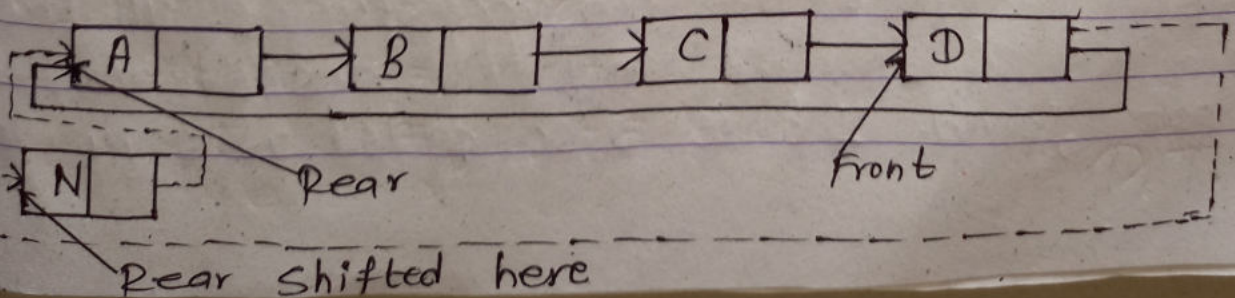
7. End


10
Rear


Front
10
20
Front
Rear


10
20
30
Front
Rear


10
20
30
40
front
Rear


A
B
C
D
N
Rear
Front
Rear shifted here

* Dequeue Algorithm:

Deleting last node of circular linked list act as dequeue operation of linked list implementation of circular queue.

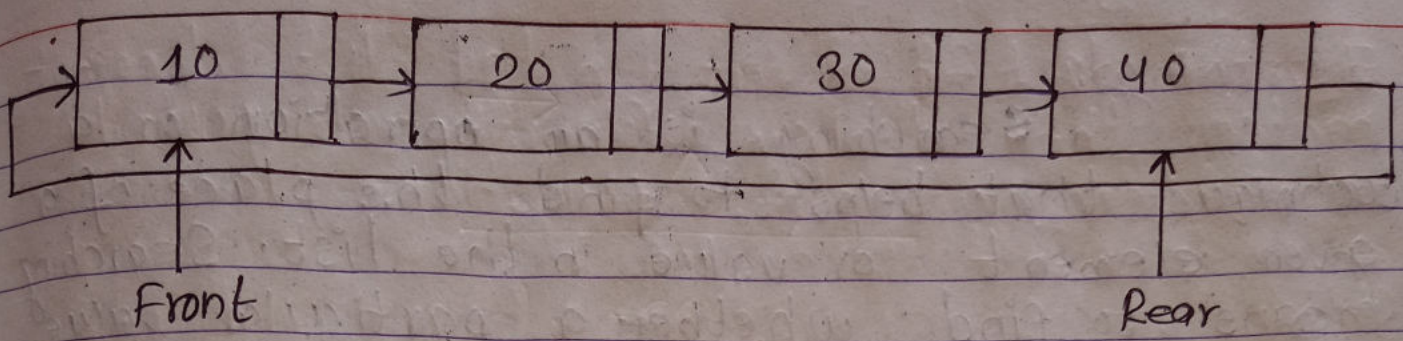
1. Start

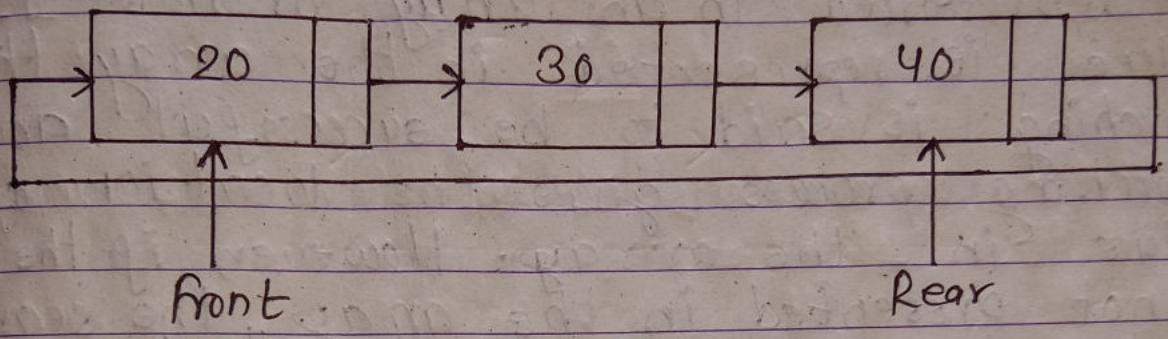
2. if first == null then
   Print "empty list" and exit

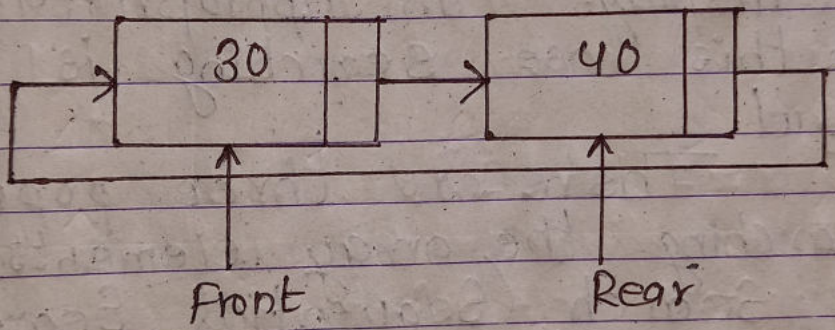
3. else if first == last
   - Print "first.info" as deleted element
   - Set, first = last = null.

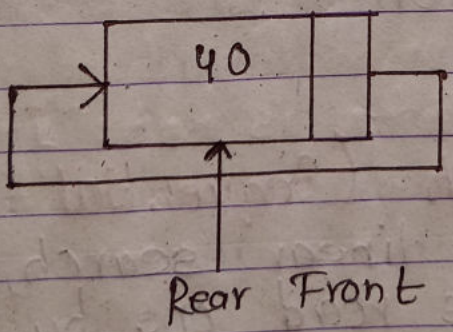
4. else
   Set, temp = first
   while (temp → next != last)
   Set, temp = temp → next
   End while
   Print the deleted element = last → info
   Set, last = temp.
   Set, last → next = first

5. Stop


10
20
30
40
Front
Rear


20
30
40
Front
Rear


30
40
Front
Rear


40
Rear
Front